Principios de Diseño de Smalltalk

por Daniel H H Ingalls

(Publicado originalmente en Byte, Agosto de 1981, dedicada a Smalltalk)

El propósito del proyecto Smalltalk es proveer soporte de computación al espíritu creativo que todos llevamos dentro. Nuestro trabajo se desarrolla a partir de una visión que incluye a un individuo creativo y al mejor hardware de computación disponible. Elegimos concentrarnos en dos áreas principales de investigación: un lenguaje de descripción (lenguaje de programación) que sirve de interfaz entre los modelos en la mente humana y aquellos en el hardware, y un lenguaje de interacción (interfaz al usuario) que adapte el sistema de comunicación humano a la computadora. Nuestro trabajo ha seguido un ciclo de dos a cuatro años que se puede ver como paralelo al método científico:

· Construir un programa de aplicación dentro del sistema actual (hacer una observación)

· Basado en esta experiencia, rediseñar el lenguaje (construir una teoría)

· Construir un nuevo sistema basado en el nuevo diseño (hacer una predicción que puede ser corroborada)

El sistema Smalltalk-80 marca nuestra quinta iteración sobre este ciclo. En este artículo, presento algunos de los principios generales que hemos observado en el curso de nuestro trabajo. Si bien la presentación se basa frecuentemente en Smalltalk, los principios son más generales y deberían ser útiles para evaluar otros sistemas y guiar trabajo futuro.

Para ir empezando, un principio que es más social que técnico y que es largamente responsable de la particular orientación del proyecto Smalltalk:

Dominio Personal: *Si un sistema es para servir al espíritu creativo, debe ser completamente entendible para un individuo solitario.*

El punto aquí es que el potencial humano se manifiesta en los individuos. Para efectivizar este potencial, debemos proveer un medio que pueda ser dominado completamente por un individuo. Cualquier barrera que exista entre el usuario y alguna parte del sistema será finalmente una barrera a la expresión creativa. Cualquier parte del sistema que no pueda ser cambiada, o que no es lo suficientemente general es probablemente un origen de impedimentos. Si una parte del sistema funciona de manera diferente del resto, esa parte requiere un esfuerzo adicional para controlarla. Esa complicación añadida puede afectar el resultado final, e inhibir futuros esfuerzos en esa área. Podemos entonces inferir un principio general de diseño:

Buen Diseño: *Un sistema debería ser construido con un mínimo conjunto de partes no modificables; esas partes debieran ser tan generales como sea posible; y todas las partes del sistema deberían estar mantenidas en un esquema uniforme.*

Lenguaje

Al diseñar un lenguaje para ser usado con computadoras, no es necesario mirar muy lejos para buscar indicaciones útiles. Todo lo que sabemos sobre cómo la gente piensa y se comunica es aplicable. Los mecanismos del pensamiento y la comunicación humanos han sido pulidos durante millones de años, y deberíamos considerarlos bien diseñados. Más aún, como deberemos trabajar con este diseño durante el

próximo millón de años, nos ahorrará tiempo si hacemos que nuestros modelos de computación sean compatibles con la mente, en vez de hacerlo al revés.

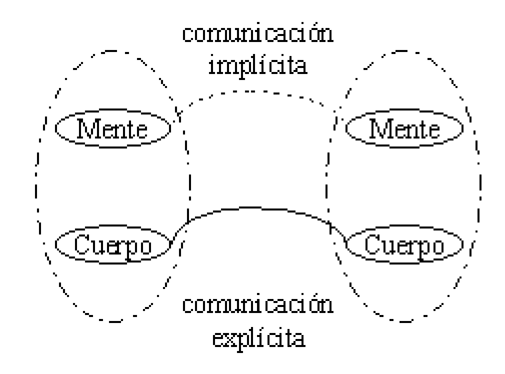


Figura 1: El alcance del diseño de un lenguaje.

La comunicación entre dos personas (o entre una persona y una computadora) incluye comunicación en dos niveles. La comunicación explícita incluye la información que es transmitida en determinado mensaje. La comunicación implícita incluye las suposiciones relevantes comunes a los dos seres.

La figura 1 ilustra los principales componentes en nuestra discusión. Una persona es presentada como teniendo un cuerpo y una mente. El cuerpo es el lugar de la experiencia primaria, y en el contexto de esta discusión, es el canal físico a través del cual el universo es percibido y a través del cual las intenciones son realizadas. La experiencia es recordada y procesada en la mente. El pensamiento creativo (sin internarnos en su funcionamiento) puede ser visto como la aparición espontánea de información en la mente. El lenguaje es la clave a esta información:

Propósito del lenguaje: *Proveer un esquema para la comunicación.*

La interacción entre dos individuos es representada en la figura 1 como dos arcos. El arco sólido representa la comunicación explícita: las palabras y gestos expresados y percibidos. El arco a rayas representa la comunicación implícita: la cultura compartida y experiencia que forma el contexto para la comunicación explícita. En la interacción humana, mucho de la comunicación real se realiza por referencias a un contexto compartido, y el lenguaje humano está construido sobre estas alusiones. Con las computadoras pasa lo mismo.

No es coincidencia que una computadora pueda ser vista como uno de los participantes de la figura 1. En este caso, el "cuerpo" provee una pantalla visual de información, y percepción de la entrada del usuario humano. La "mente" de una computadora incluye los elementos de memoria y procesamiento internos y sus contenidos. La figura 1 muestra que hay varios ítems involucrados en el diseño de un lenguaje de computadora:

Alcance: *El diseño de un lenguaje para usar computadoras debe tratar con modelos internos, medios externos, y con la interacción entre ellos tanto en el humano como en la computadora.*

Este hecho es responsable de la dificultad de enseñar Smalltalk a gente que ve a los lenguajes de computadora en un sentido más restringido. Smalltalk no es simplemente una mejor manera de organizar procedimientos o un técnica distinta para administración de memoria. No es sólo un jerarquía extensible de tipos de datos, o una interfaz gráfica al usuario. Es todas estas cosas, y cualquier otra que sea necesaria para soportar las interacciones ilustradas en la figura 1.

Objetos que se Comunican

La mente observa un vasto universo de experiencia, tanto inmediata como recordada. Uno puede tener un sentido de unidad con el universo simplemente al dejar que esta experiencia sea, simplemente como es. Sin embargo, si uno quiere participar, literalmente tomar parte en el universo, uno debe diferenciar. Al hacerlo uno identifica un objeto en el universo, y simultáneamente todo el resto se convierte en no-ese-objeto. La diferenciación por sí misma es un comienzo, pero el proceso de distinguir no es sencillo. Cada vez que quieras hablar sobre "esa silla que está ahí" tienes que repetir el proceso entero de diferenciarla. Acá es donde el acto de referenciar entra: podemos asociar un identificador único al objeto, y, desde ese momento, sólo el mencionar al identificador es necesario para referenciar al objeto original.

Hemos dicho que un sistema de computación debe proveer modelos que sean compatibles con los de la mente. Por lo tanto:

Objetos: *Un lenguaje de computación debe soportar el concepto de "objeto" y proveer una manera uniforme de referirse a los objetos de universo.*

El administrador de almacenamiento de Smalltalk provee un modelo orientado a objetos de la memoria de todo el sistema. La referenciación uniforme se obtiene simplemente asociando un entero que no se repite a cada objeto del sistema. Esta uniformidad es importante porque significa que las variables en el sistema pueden indicar valores ampliamente variados y aún así ser implementadas como simples posiciones de memoria. Se crean objetos al evaluarse las expresiones, y pueden ser pasados de un lado a otro gracias a la referenciación uniforme, por eso no es necesaria una manera de almacenamiento en los procedimientos que los manipulan. Cuando todas las referencias a un objeto han desaparecido del sistema, el propio objeto se esfuma, y su espacio de almacenamiento es recuperado. Este comportamiento es esencial para soportar completamente la metáfora de objetos.

Administración del Almacenamiento: *Para ser auténticamente "Orientado a Objetos" un sistema debe proveer administración automática del almacenamiento.*

Una manera de ver si un lenguaje está funcionando bien es ver si los programas parecen estar haciendo lo que hacen. Si están salpicados con instrucciones de administración del almacenamiento, entonces su modelo interno no está bien adaptado al del los humanos. ¿Podrías imaginar tener que preparar a alguien para cada cosa que le digas, o tener que informarle cuando terminaste con determinado tópico y ya puede ser olvidado?

Cada objeto en nuestro universo tiene su propia vida. Similarmente, el cerebro provee procesamiento independiente juntamente con el almacenamiento de cada objeto mental. Esto sugiere un tercer principio para el diseño orientado a objetos:

Mensajes: *La computación debería ser vista como una capacidad intrínseca de los objetos que pueden ser invocados uniformemente enviándoles mensajes.*

Así como los programas se ensucian si el almacenamiento de los objetos debe ser tratado explícitamente, el control en el sistema se vuelve complicado si debe ser tratado extrínsicamente. Consideremos el proceso de sumarle 5 a un número. En la mayoría de los sistemas de computación, el compilador averigua que tipo de número es y genera el código para sumarle 5. Esto no está bien para un sistema orientado a objetos porque los tipos exactos de números no pueden ser determinados por el compilador (más sobre esto después). Una solución posible es llamar a una rutina general de suma que examine el tipo de los argumentos para determinar la acción apropiada. Este no es un buen enfoque porque significa que esta rutina crítica debe ser editada por novatos que sólo quieren experimentar con sus propias clases de números. Es también un diseño pobre porque los aspectos internos de los objetos están desparramados por todo el sistema.

Smalltalk provee una solución mucho más limpia: Envía el nombre de la operación deseada, juntamente con cualquier parámetro necesario, como un mensaje al número, entendiendo que el receptor es el que mejor sabe cómo realizar la operación deseada. En vez de un procesador pulverizador de bits destripando y saqueando estructuras de datos, tenemos un universo de objetos que se portan bien, que cortésmente se piden unos a otros realizar sus variados deseos. La transmisión de mensajes es el único proceso que se hace afuera de los objetos y así es como debiera ser, ya que los mensajes viajan entre objetos. El principio de buen diseño puede ser reexpresado para lenguajes:

Metáfora Uniforme: *Un lenguaje debería ser diseñado alrededor de una metáfora poderosa que pueda ser aplicada uniformemente en todas las áreas.*

Ejemplos de éxitos en esta área incluyen a LISP, que está construido sobre el modelo de listas enlazadas; APL, que está construido sobre el modelo de arreglos; y Smalltalk, que está construido sobre el modelo de objetos que se comunican. En cada caso, las aplicaciones grandes son vistas en la misma manera en que las unidades fundamentales de las cuales el sistema está construido. Especialmente en Smalltalk, la interacción entre los objetos más primitivos es vista en la misma manera que la interacción de más alto nivel entre la computadora y su usuario. Todo objeto en Smalltalk, hasta un humilde entero, tiene un conjunto de mensajes, un protocolo, que define la comunicación explícita a la que ese objeto puede responder. Internamente, los objetos pueden tener almacenamiento local y acceso a otra información compartida que comprenden el contexto implícito de toda comunicación. Por ejemplo, el mensaje + 5 (sume cinco) lleva una suposición implícita de que el sumando es el número que recibe el mensaje.

Organización

Una metáfora uniforme provee el marco en el cual se pueden construir sistemas complejos. Algunos principios de organización relacionados entre sí contribuyen a la administración exitosa de la complejidad. Para empezar:

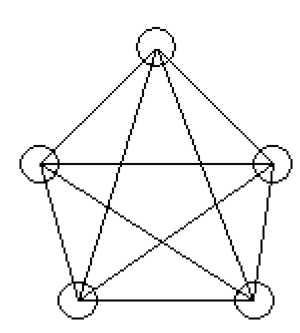


Figura 2: Complejidad de un sistema.

Al incrementarse la cantidad de componentes de un sistema, la probabilidad de interacción no deseada crece rápidamente. Por esto, un lenguaje de computadora debería ser diseñado para minimizar las posibilidades de esa interdependencia.

Modularidad: *Ningún componente en un sistema complejo debería depender de los detalles internos de ningún otro componente.*

Este principio es ilustrado en la figura 2. Si hay N componentes en un sistema, entonces hay aproximadamente N2 dependencias potenciales entre ellos. Si los sistemas de computación son para alguna vez asistir a las tareas humanas complejas, deben ser diseñados para minimizar esta interdependencia. La metáfora de envío de mensajes provee modularidad al desacoplar la intención del mensaje (incorporado en el nombre) del método usado por el receptor para realizar la intención. La información estructural es similarmente protegida porque todo acceso al estado interno de un objeto es a través de esta misma interfaz de mensajes.

La complejidad de un sistema puede muchas veces ser reducida agrupando componentes similares. Este agrupamiento es conseguido a través del tipado de datos en los lenguajes de programación convencionales, y a través de clases en Smalltalk. Una clase describe otros objetos -su estado interno, el protocolo de mensajes que reconocen, y los métodos internos para responder a esos mensajes. Los objetos así descriptos se llaman instancias de la clase. Incluso las propias clases entran en este esquema; simplemente son instancias de la clase Class, que describe el protocolo y la implementación apropiados para la descripción de los objetos:

Clasificación: *Un lenguaje debe proveer un medio para clasificar objetos similares, y para agregar nuevas clases de objetos en pie de igualdad con las clases centrales del sistema.*

La clasificación es la objetivación de la idadidad. En otras palabras, cuando un humano ve una silla, la experiencia es interpretada tanto literalmente como "es mismísima silla" y en forma abstracta cono "esa cosa similar a una silla". Esta abstracción resulta de la maravillosa capacidad de la mente de combinar experiencias "similares", y esta abstracción se manifiesta como otro objeto en la mente, la silla Platónica o silla*cidad*.

Las clases son el principal mecanismo de extensión en Smalltalk. Por ejemplo, un sistema de música sería creado agregando nuevas clases que describen la representación y el protocolo de interacción de Note (nota), Melody (melodía), Score (partitura), Timbre, Player (ejecutante), etc. La cláusula "en pie de igualdad" del principio recién enunciado es importante porque asegura que el sistema va a ser usado como fue diseñado. En otras palabras, una melodía puede ser representada como una colección ad hoc de Integers (números enteros) que representan altura, duración y otros parámetros, pero si el lenguaje permite manejar notas tan fácilmente como enteros, entonces el usuario va a describir naturalmente una melodía como una colección de notas. En cada etapa del diseño, un humano va a elegir en forma natural la representación más efectiva si el sistema la provee. El principio de modularidad tiene una implicación interesante para los componentes procedurales de un sistema.

Polimorfismo: *Un programa sólo debería especificar el comportamiento esperado de los objetos, no su representación.*

Un afirmación inferida de este principio es que un programa nunca debería declarar que cierto objeto es un SmallInteger (entero chico) o un LargeInteger (entero grande), sino sólo que responde al protocolo de los enteros. Esta descripción genérica es crucial para los modelos del mundo real.

Considera una simulación de tránsito automotor. Muchos procedimientos en este sistema se referirán a los diversos vehículos involucrados. Supón que uno quisiera agregar, por ejemplo, un camión barrecalles. Una cantidad substancial de computación (por recompilación) y posibles errores estarían involucrados para hacer esta simple extensión si el código dependiera de los objetos que manipula. La interfaz de mensajes establece un marco ideal para esta extensión. Dado que los camiones barrecalles soportan el mismo protocolo que los demás vehículos, no hacen falta cambios para incluirlos en la simulación.

Factorización: *Cada componente independiente de un sistema sólo debería aparecer en un sólo lugar.*

Hay muchas razones para este principio. Primero que todo, ahorra tiempo, esfuerzo y espacio si los agregados al sistema sólo necesitan hacerse en un lugar. Segundo, los usuarios pueden encontrar más fácilmente un componente que satisfaga una dada necesidad. Tercero, en la ausencia de una factorización apropiada, aparecen problemas para sincronizar cambios y para asegurar que todos los componentes interdependientes son consistentes. Puedes ver que una falla en la factorización implica una violación a la modularidad.

Smalltalk promueve diseños bien factorizados a través de la herencia. Todas las clases heredan comportamiento de su superclase. Esta herencia se desarrolla a través de clases cada vez más generales, terminando finalmente con la clase Object (objeto) que describe el comportamiento mínimo de todos los objetos del sistema. En nuestra simulación del tránsito, StreetSweeper (camión barrecalles), y todas las otras clases de vehículos, serían descriptas como subclases de una clase Vehicle (vehículo) general; así heredando comportamiento mínimo apropiado y evitando la repetición de los mismos conceptos en muchos lugares distintos. La herencia ilustra una forma más pragmática de factorización:

Reaprovechamiento: *Cuando un sistema está bien factorizado, un gran reaprovechamiento está disponible tanto para los usuarios como para los implementadores.*

Toma el ejemplo de ordenar una secuencia de objetos. En Smalltalk, el usuario definiría un método sort (ordenar) en la clase OrderedCollection (secuencia). Cuando esto ya ha sido hecho, todas las formas de secuencias en el sistema van a adquirir esta nueva capacidad a través de la herencia. Por otra parte, vale notar que el mismo método sirve tanto para alfabetizar texto como para ordenar números, ya que el protocolo de comparación es reconocido tanto por las clases de texto como por las de números.

Los beneficios de la estructuración para los implementadores son obvios. Para empezar, habrá menos primitivas para implementar. Por ejemplo, todos los gráficos en Smalltalk se hacen con una sola

operación primitiva. Con sólo una tarea para hacer, un implementador puede aplicar su máxima atención a cada instrucción, sabiendo que cada pequeña mejora en eficiencia será amplificada en todo el sistema. Es natural preguntar qué conjunto de operaciones primitivas serán suficientes para soportar un sistema de computación entero. La respuesta a esta pregunta se llama especificación de una máquina virtual.

Máquina Virtual: *Una especificación de máquina virtual establece un marco para la aplicación de tecnología.*

La máquina virtual de Smalltalk establece un modelo orientado a objetos para el almacenamiento, un modelo orientado a mensajes para el procesamiento, y un modelo de bitmap (mapa de bits) para el despliegue visual de información. A través del uso de microcódigo, y eventualmente hardware, la performance del sistema puede ser mejorada dramáticamente sin ningún compromiso de las otras virtudes del sistema.

Interfaz al Usuario

Una interfaz al usuario es simplemente un lenguaje en el que la mayor parte de la comunicación es visual. Dado que la presentación visual se asimila mucho a la cultura humana establecida, la estética juega un rol muy importante en esta área. Como toda la capacidad de un sistema de computación finalmente es entregada a través de la interfaz al usuario, la flexibilidad es esencial también acá. Una condición habilitante para la flexibilidad adecuada de una interfaz al usuario puede ser enunciada como un principio orientado a objetos:

Principio Reactivo: *Cada componente accesible al usuario debería ser capaz de presentarse de una manera entendible para ser observado y manipulado.*

Este criterio está bien soportado por el modelo de objetos que se comunican. Por definición, cada objeto provee un protocolo de mensajes apropiado para la interacción. Este protocolo es esencialmente un microlenguaje particular para sólo ese tipo de objeto. Al nivel de la interfaz al usuario, el lenguaje apropiado para cada objeto en la pantalla es presentado visualmente (como texto, menúes, imágenes) y percibido a través de la actividad del teclado y el uso de un dispositivo apuntador.

Debería notarse que los sistemas operativos parecen violar este principio. Aquí el programador debe alejarse de un marco de descripción que de otra manera es consistente, dejar cualquier contexto ya construido, y enfrentarse con un entorno completamente distinto y usualmente muy primitivo. Esto no tiene por qué ser así:

Sistema Operativo: *Un sistema operativo es una colección de cosas que no encajan dentro de un lenguaje. No debería existir.*

Aquí hay algunos ejemplos de componentes de sistemas operativos convencionales que han sido incorporados naturalmente al lenguaje Smalltalk:

· Administración del Almacenamiento - Enteramente automático. Los objetos son creados por un mensaje a su clase y destruidos cuando no nadie los referencie. La expansión del espacio de direccionamiento mediante la memoria virtual es igualmente transparente.

· Sistema de Archivos - Está incorporado al esquema usual a través de objetos como Files (archivos) y Directories (directorios) con protocolos de mensajes que soportan el acceso a archivos.

· Manejo de la Pantalla - La pantalla es simplemente una instancia de la clase Form (figura), que es continuamente visible, y los mensajes de manipulación gráfica definidos en esa clase se usan para cambiar la imagen visible.

· Entrada del Teclado - Los dispositivos de entrada del usuario se modelan similarmente como objetos con mensajes apropiados para determinar su estado o leer la su historia como una secuencia de eventos.

· Acceso a Subsistemas - Los subsistemas se incorporan naturalmente como objetos independientes dentro de Smalltalk: aquí pueden usar el amplio universo de descripción existente, y aquellos que involucran interacción con el usuario pueden participar como componentes de la interfaz al usuario.

· Debugger (herramienta de depuración) - El estado del procesador de Smalltalk es accesible como una instancia de la clase Process (proceso) que es dueña de una cadena de stacks. El debugger es sólo un subsistema de Smalltalk que tiene acceso a manipular el estado de un proceso suspendido. Es de notar que casi el único error de tiempo de ejecución que puede ocurrir en Smalltalk es que un mensaje no sea entendido por su receptor.

Smalltalk no tiene "sistema operativo" como tal. Las operaciones primitivas necesarias, como leer una página del disco, son incorporadas como métodos primitivos en respuesta a mensajes Smalltalk normales.

Trabajo futuro

Como es de esperar, hay trabajo por hacer en Smalltalk. La parte mas fácil de describir es la aplicación continuada de los principios de este artículo. Por ejemplo, el sistema Smalltalk-80 es deficiente en su factorización porque sólo soporta herencia jerárquica. Futuros sistemas Smalltalk generalizarán este modelo a herencia arbitraria (múltiple). Además, el protocolo de mensajes no se ha formalizado. La organización provee protocolos, pero actualmente es sólo una cuestión de estilo que los protocolos sean consistentes de una clase a otra. Esto puede ser remediado fácilmente proveyendo objetos protocolo apropiados que puedan ser compartidos consistentemente. Esto posibilitará el tipado formal de variables por protocolo, sin perder las ventajas del polimorfismo.

El otro trabajo que queda es menos fácil de articular. Hay claramente otros aspectos del pensamiento humano que no han sido tratados en este artículo. Deben ser identificados como metáforas que puedan complementar los modelos existentes del lenguaje.

A veces el avance en los sistemas de computación parece deprimentemente lento. Nos olvidamos que las máquinas de vapor eran "high-tech" para nuestros abuelos. Soy optimista sobre la situación. Los sistemas de computación están, realmente, simplificándose y, como resultado, más usables. Quisiera cerrar con un principio general que gobierna este proceso:

Selección Natural: *Los lenguajes y sistemas que son de buen diseño persistirán, sólo para ser reemplazados por otros mejores.*

Así como el reloj avanza, mejor y mejor soporte de computación para el espíritu creativo está evolucionando. La ayuda está en camino.